Algorithms Exam
TIN093/DIT093/DIT602

Course: Algorithms

Course code: TIN 093 (CTH), DIT 093 and DIT 602 (GU)

Date, time: 26th October 2022, 8:30–12:30

Place: Johanneberg

Responsible teacher: Peter Damaschke, Tel. 5405, email `ptr@chalmers.se`

Examiner: Birgit Grohe

Exam aids: dictionary,
printouts of the Lecture Notes (possibly with own annotations),
printouts of the Assignments (possibly with own annotations of solutions),
one additional A4 paper (both sides) with own handwritten notes.

Time for questions: around 9:30 and around 11:00.

Solutions: will be published after the exam.

Results: will appear in ladok.

Point limits: 28 for 3, 38 for 4, 48 for 5; PhD students: 38. Maximum: 60.

Inspection of grading (exam review): to be announced.

Instructions and Advice:

- First read through all problems, such that you know what is unclear to you and what to ask the responsible teacher.
- Write solutions in English.
- Start every new problem on a new sheet of paper.
- Write your exam number on every sheet.
- Write legible. Unreadable solutions will not get points.
- Answer precisely and to the point, without digressions.
  Unnecessary additional writing does not only cost time.
  It may also obscure the actual solutions.
- But motivate all claims and answers.
- Strictly avoid code for describing a complex algorithm.
  Instead *explain* in your words how the algorithm works.
- If you cannot manage a problem completely, still provide your approach or partial solution to earn some points.
- Facts from the course material can be assumed to be known.
  You don't have to repeat their proofs.

Remark: The number of points is not always "proportional" to the length or difficulty of a solution, but it may also be influenced by the importance of the topics and skills.

Good luck!

Problem 1 (10 points)

Imagine that n heavy boxes are lined up. They have identity numbers from 1 to n, but somehow the ordering got confused, and we want to arrange the boxes from left to right according to their identity numbers $1, \ldots, n$. But in every step, *only two neighbored boxes* can be chosen and *swapped* (because they are heavy, moreover, there is not enough extra space for temporal storage of boxes). We want to sort the boxes by using a minimum number of swaps.

Example of such a sequence of swaps:
$2431 \to 2341 \to 2314 \to 2134 \to 1234$.

We propose a simple greedy algorithm: Choose any two neighbored boxes such that the left box has a larger identity number than the right box, and swap them. Iterate this step until the ordering $1, \ldots, n$ is established.
Actually, this is the well-known Bubblesort algorithm, but considered from a different angle: We want to prove that this algorithm minimizes the number of swaps, for every fixed input.
Recall that an inversion is a pair of numbers $i < j$ such that i is located to the right of j. Let inv denote the number of inversions in the initially given ordering. (In the example above, we have $inv = 4$.)
Prove the following two statements (from which the optimality of the number of swaps easily follows):

1.1. Every sequence of swaps that finally creates the ordering $1, \ldots, n$ needs at least inv swaps. (5 points)

1.2. The proposed algorithm performs exactly inv swaps. (5 points)

Make sure that your explanations are strict, conclusive, and complete.

Problem 2 (16 points)

An automatic system for train journey planning has to solve the following type of problem: Given a place A of departure, a destination B, and a desired (latest) arrival time at B, find a journey with the latest possible departure time at A.
In order to design such a system we model the stations by nodes of a graph, and the existing train rides by directed edges (C,D) annotated with departure and arrival times. Note that there can exist many directed edges between the same nodes C and D, with different times.
We assume the following restriction: When a user wants to travel from A to B, the system allows only train rides (C,D) where D is closer to the final destination B than C is. In other words, the geographical distance from D to B must be smaller than from C to B. (In general it can be beneficial to take a train to a place farther away from B, e.g., in order to catch a faster train there. But for simplicity we ignore this possibility here.) All pairwise distances are already known, i.e., precomputed. We also assume that all trains are on time, i.e., no delays occur.
For every station X, we naturally define OPT(X) to be the latest possible departure time at X such that B can still be reached before the desired time.

Describe how you would compute a connection with the latest possible departure time OPT(A). You can omit the details of backtracing. Argue why your algorithm is correct and works in polynomial time in the size of the graph. (But you need not state a specific time bound.) Also point out how your algorithm uses the mentioned restriction that every train must bring the passenger closer to B.

Advice: Do not spend too much effort on creating a "nice formula" for computing the OPT-values. It is probably more convenient here to describe the computation in words, yet clearly and unambiguously.

Problem 3 (10 points)

In statistical surveys one often has to compute so-called percentiles with equal step lengths. In an abstract formulation: We are given a huge set of n numbers, and some integer $k < n$. We want to know, for all $i = 1, \ldots, k$, the element of rank $\lfloor (i/k)n \rfloor$ in this set. (Remember that the "element of rank r" is defined to be the rth smallest element.)
One may sort the set in $O(n \log n)$ time and then simply read off the results. Instead we may also apply some $O(n)$-time algorithm for the Selection problem k times, which needs $O(kn)$ time in total. But we can do better than in both of these obvious approaches:

Give an algorithm that solves this problem already in $O(n \log k)$ time.

Advice: Apply divide-and-conquer from scratch. Do not use the "master" theorem to figure out the time bound; here it is more convenient to compute the time in an ad-hoc way. Still you may use, without proof, the known fact that *one* element with any specific rank can always be found in $O(n)$ time.

Problem 4 (12 points)

In the Set Packing problem we are given a family of subsets of a finite set U, and an integer k, and the problem is to find k pairwise disjoint subsets in this family, if they exist. Set Packing is NP-complete. In a restricted case of this problem that we call Uniform Set Packing, all subsets in the given family have equal size. Show that Uniform Set Packing is still NP-complete. In detail:

4.1. Show that the problem is in NP. (2 points)

4.2. Describe a polynomial-time reduction. (6 points)

4.3. For your reduction proposed in 4.2, prove equivalence of the given and the constructed instance. (4 points)

Hint to 4.2 (maybe obvious): Do a reduction from Set Packing, by adding appropriate new elements that might also be outside U.

Problem 5 (12 points)

We are given a directed graph $G = (V, E)$ with n nodes and m edges, and some start node $s \in V$. The problem is to find a directed path that starts in s and never ends (although the graph is finite!). More precisely, the solution shall consist of a directed path P that starts and s and leads to some directed cycle C. Then we can traverse P once, and then traverse C in principle infinitely often. The path P is allowed to be empty (have zero edges), which is equivalent to $s \in C$, or to be non-empty.

Give an algorithm that outputs such a path P and cycle C, or reports that no solution exists in G. Explain why your algorithm is correct. Moreover, it should run in $O(n+m)$ time. You can (and should) use standard algorithms known from the course material as building blocks, without describing their internal details again. Describe only how you put them together.

Solutions (attached after the exam)

1.1. Consider any two neighbored boxes with numbers j (left box) and i (right box), where $i < j$. This is an inversion, which disappears by swapping these two boxes. Any other box that was to the left (to the right) of them is still to the left (to the right) of them after the swap. Hence no other inversions are created or deleted. It follows that every swap decreases inv by only 1. Thus the necessary number of swaps is at least inv. (5 points)

1.2. The same arguments as in 1.1 show that every swap reduces inv also by at least 1. (We do not repeat them here literally. Hence, after inv swap operations, we have $inv = 0$, which means that all the boxes are now sorted. (5 points)

2. First we (temporarily) delete all edges that violate the restriction. Now the ordering by the distances to B defines a topological ordering in the remaining graph. We work on this topological ordering from behind. Trivially, OPT(B) is the desired arrival time at B. Now consider any station X. Note that OPT(Y) is already known for all stations Y that are closer to B than X is. Consider any station X that we pass. If Y is chosen to be the next station in our journey, we must arrive at Y before time OPT(Y). Therefore we choose the train from X to Y (if there is any) that arrives before OPT(Y) and has the latest departure time from X. Finally we take the maximum of all these departure times at X (for all possible successors Y) to obtain OPT(X). As soon as OPT(A) is computed, the problem instance is solved. The actual connection can then be recovered from the OPT values by backtracing. The algorithm runs in polynomial time because every directed edge (train ride) is considered at most once, and only some auxiliary computations like sorting are needed. (16 points)

3. Compute the median and split the instance in two halves, consisting of the numbers being smaller and larger, respectively, than the median. In both instances of half size, $k/2$ elements with given (and equidistant) ranks must be determined. Doing this recursively will split the given instance, for every j with $0 \leq j < \log_2 k$ into 2^{j+1} instances of equal size. At the jth recursion level we must find the median of 2^j instances of length $n/2^j$ each, which costs $O(2^j n/2^j) = O(n)$ time. Hence we need $O(n \log k)$ time in total. In the end we only have to find $O(1)$ elements with specific ranks in every instance. (10 points)

4.1. Uniform Set Packing is in NP, since we can check any given solution in polynomial time. (2 points)

4.2. For a reduction, let I be an instance of Set Packing, and let m be the maximum size of the subsets in I. We construct an instance J of Uniform Set Packing as follows: To every subset in I, say with p elements, we add $m-p$ fresh elements (that do not appear elsewhere, not even in U). Now all subsets in J have uniform size m. The number k is not changed. The reduction obviously runs in polynomial time. (6 points)

4.3. Consider any solution with k subsets from I. The corresponding subsets in J are then also pairwise disjoint, because we have added only fresh elements. The reverse direction is clear: Removing elements from disjoint sets keeps them disjoint. (4 points)

5. First we determine by BFS the set R of nodes that are reachable from s on any directed paths. Other nodes cannot be in P and C. Thus we work only in the graph H with node set R and all edges between them. We do topological sorting of H. If it succeeds, we know that H is acyclic, and no solution can exist. Suppose that topological sorting fails. Then we know that some directed cycle C exists in H, and we can find one as follows. Since all nodes in H are reachable from s, no nodes have indegree 0. Start in an arbitrary node and go backwards until some node is encountered repeatedly. At this moment we have found a cycle C. Then we run again BFS, but only on H, to find a directed path P from s to an arbitrary node of C. All algorithms involved in this procedure are known to run in $O(n+m)$ time, therefore we can omit a detailed time analysis. (12 points)